Towards lightweight state coverage

Dries Vanoverberghe
Frank Piessens

Report CW 671, August 2014

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Towards lightweight state coverage

Dries Vanoverberghe
Frank Piessens

Report CW 671, August 2014

Department of Computer Science, KU Leuven

Abstract

State coverage is a relatively new metric to evaluate the quality of test suites. While most existing test adequacy criteria measure the degree of exploration of the code under test, state coverage estimates the strength of the assertions in the test suite. Unfortunately, state coverage suffers from two important disadvantages. First, it uses a fairly complicated dependency analysis based on information flow analysis, which is both inefficient and hard to understand for end-users. Second, since state coverage focuses on incorrect assignments, it does not help to detect defects of omission, which are at least as important.

We propose Lightweight state coverage, an approximation of state coverage that only requires a simple read analysis. In addition, all reachable state which is shared between the application and the code under test must be checked using assertions.

Towards lightweight state coverage

Dries Vanoverberghe
DistriNet Research Group
Department of Computer Science
KULeuven, Belgium
Dries.Vanoverberghe@cs.kuleuven.be

Frank Piessens
DistriNet Research Group
Department of Computer Science
KULeuven, Belgium
Frank.Piessens@cs.kuleuven.be

***Abstract*—State coverage is a relatively new metric to evaluate the quality of test suites. While most existing test adequacy criteria measure the degree of exploration of the code under test, state coverage estimates the strength of the assertions in the test suite. Unfortunately, state coverage suffers from two important disadvantages. First, it uses a fairly complicated dependency analysis based on information flow analysis, which is both inefficient and hard to understand for end-users. Second, since state coverage focuses on incorrect assignments, it does not help to detect defects of omission, which are at least as important.**

We propose *Lightweight state coverage*, an approximation of state coverage that only requires a simple read analysis. In addition, all reachable state which is shared between the application and the code under test must be checked using assertions.

I. Introduction

As the role of software in society is growing constantly, the impact of software defects on the economy is huge. Currently, testing is still the most important approach to reduce the number of software defects before releasing software product. Test adequacy criteria [1] are metrics that evaluate how thorough a test suite evaluates the product. In this way, they help to decide whether a product is sufficiently tested, and they provide feedback in which area the test suite can be improve.

Due to their simplicity and efficiency, structural metrics such as statement coverage or branch coverage are commonly used as part of the standard build cycle. While there is a variety of more advanced metrics, all of the well-established metrics essentially focus on the degree of exploration of the code. They do not measure the strength of the test oracle, the properties that must be satisfied by the code. This is not surprising since it is hard to check completeness of the test oracle when there is no complete set of formal requirements available which is usually the case in practice.

State coverage [2]–[5] is a relatively new metric that aims to evaluate the strength of the test oracle. State coverage is based on the hypothesis that updates to the execution state which are not validated by the test oracle are more likely to cause software defects. Existing experiments have shown that state coverage [2]–[4] is a promising metric to augment structural coverage metrics. While structural metrics raise the question whether a particular statement is reachable, state coverage questions which invariant is established by an update to the execution state.

Unfortunately, state coverage suffers from two important disadvantages. First, state coverage requires a complicated dependency analysis to decide which state updates are checked by an assertion. This dependency analysis is highly related to information flow analysis. In some sense, each assignment in the program is a source of information. During the execution of the program, this information is transformed. Assertions can be modeled as sinks of information, thus information flow analysis results in an over-approximation of the assignments that influence the assertions. Since most algorithms for information flow analysis are highly complex, the dependency analysis results in a high runtime overhead.

Second, state coverage has a direct focus on incorrect assignments. In practice, a large fraction of software defects is caused by missing logic rather than incorrect logic [6]–[8]. Since this implies missing rather than incorrect assignments, state coverage does not help to this category of defects. This is one potential explanation why recent experiments [5] failed to show a correlation between the state coverage of a test suite, and its number of detected defects.

In this paper, we propose *Lightweight state coverage*, a variant of state coverage that alleviates these problems. First, lightweight state coverage simplifies the dependency analysis. In stead of dynamic taint tracking, lightweight state coverage only tracks which fields are read while executing the body of an assertion. As a result, the set of assignments that influence an assertion is a more coarse over-approximation. However, since information flow analysis is undecidable in general, any algorithm always results in an approximation.

Secondly, lightweight state coverage requires that all reachable state after each transition in the code under test must be validated using assertions. Unlike Koster et. al. [3], this approach is object sensitive [4]. Unlike the standard object sensitive approach [4], [5], all reachable state must be checked rather than the state which has been redefined during the execution of the code under test. Therefore, lightweight state coverage is stricter and will require a stronger set of assertions to achieve good coverage.

To evaluate whether runtime overhead of lightweight state coverage, we have performed a case study using the Apache Commons Collections library [9]. This preliminary evaluation indicates that the execution time maximally increases by a factor 10, which is significant improvement over Koster et. al. [3].

The remainder of this paper is structured as follows. First, we motivate the necessity of lightweight state coverage based on an example (Section II). Next, Section III defines lightweight state coverage and contains an efficient algorithm

```
1  class Set {
2    Node head;
3    int count;
4
5    Set() {
6      head = new Node(0);
7      count = 0;
8    }
9    void add(int value) {
10     Node prev = locate(value);
11     Node cur = prev.next;
12     if (cur == null || value < cur.value) {
13       prev.next = new Node(value);
14       prev.next.next = cur;
15     }
16     count++;
17   }
18   boolean contains(int value) {
19     Node prev = locate(value);
20     Node cur = prev.next;
21     return cur != null && cur.value == value;
22   }
23   Node locate(int value) {
24     Node prev = head;
25     Node cur = head.next;
26     while (cur != null && cur.value < value) {
27       prev = cur;
28       cur = prev.next;
29     }
30     return prev;
31   }
32 }
33 class Node {
34   Node next;
35   int value;
36   Node(int value) {
37     this.value = value;
38   }
39 }
40 class SetTests {
41   void testAdd() {
42     Set s = new Set();
43     s.add(0); s.add(1);
44     assertTrue(s.contains(0));
45     assertTrue(s.contains(1));
46     assertFalse(s.contains(2));
47   }
48   void testAdd2() {
49     Set s = new Set();
50     s.add(0); s.add(0);
51     assertTrue(s.contains(0));
52     assertFalse(s.contains(1));
53   }
54 }
```

Fig. 1. A minimal implementation of a set data structure

to compute it. Then, Section IV presents a preliminary evaluation and Section V describes the relation between lightweight state coverage and existing work. Finally, we present the conclusion in Section VI.

II. Motivating example

Figure 1 shows a minimal implementation of a set based on linked lists [1]. It contains the constructor *Set* to create an empty set, and two methods *add* and *contains* to add an element to the set and check set membership respectively.

Test adequacy criteria have two advantages. First, they help to decide whether a piece of code has been sufficiently tested. In this way, it is possible to reduce the number of defects discovered after product release, which significantly reduces their cost. For example, the test case *testAdd* in Figure 1 only executes five of the seven lines of code of the *add* method, which is rather low given the small size of the program. Second, test adequacy criteria help to identify the parts of the program where additional testing is most useful. For example, in order to improve the test suite, one needs to create a test case which executes line 13 and 14 of the method *add* (For example, *testAdd2*).

To keep this example simple, we have used line coverage to measure the strength of the test suite. However, there is a variety of metrics which all to some degree measure the same property: the *degree of exploration*, the fraction of execution paths which have been explored. If we focus on the *add* method in isolation, there are essentially two equivalent paths: Either the element is already in the set, in which case the variable *cur* is non-null and its *value* field equals the element we want to insert, or it is not already in the set. Therefore, the test suite covers all isolated paths of the method *add* and line coverage and path coverage coincide. In general, this is not the case, for example the method *locate* has full line coverage, but has an infinite number of execution paths.

Regardless of the fact that all execution paths of the method *add* have been executed by the test suite, the code has a defect. When we add the same element twice, the *count* field of the set is no longer the same as the number of its elements. This illustrates a shortcoming of metrics which only evaluate the degree of exploration: it is not sufficient to execute the code, one has to check that the code realizes the requirements. In the absence of assertions, the test suite only validates that the code does not crash. Unfortunately, it is much harder to evaluate whether the test oracle, the properties which are checked by the test suite, is complete with respect to the requirements.

State coverage is a relatively new metric to evaluate the strength of the test oracle. It measures the fraction of state updates which have been validated inside an assertion. For the example, state coverage detects that the *count* field is updated in the method *add*, but the updated value is not checked by assertions. Therefore, it suggests that a new assertion must be added to validate the *count* field in both test cases.

Unfortunately, an empirical evaluation of state coverage [5] failed to show that there is positive correlation between state coverage and the number of detected defects in a program. One potential explanation is that state coverage focuses on incorrect assignment statements and therefore does not help in detecting *errors of omission*, which represent a large fraction of software defects [6], [7], especially amongst persistent defects. For example, suppose the update to the count field at line 16 was simply missing rather than incorrect, then the test suite may still have full state coverage.

To alleviate this, lightweight state coverage requires that all reachable fields must be checked by an assertion, rather than all assignments to a field. Since the *count* field is reachable, the test case should add an assertion to check correctness in order to gain full lightweight state coverage.

In addition, lightweight state coverage only needs assertions for reachable fields, thus it is not necessary to add assertions for intermediate state which becomes unreachable after a computation has finished.

[1] This example is a simplified version of a defect which was discovered using state coverage [4]

III. LIGHTWEIGHT STATE COVERAGE

In this section, we define *Lightweight state coverage*, a novel approach to asses the quality of assertions in a test suite. Then, we provide a simple algorithm to collect lightweight state coverage during the execution of a test suite.

A. Definition

In general, lightweight state coverage makes a distinction between two domains in the code. First, the code under test is the program that must be evaluated. Second, the test suite including all auxiliary methods that are used to invoke the program and check its correctness are part of the test code.

Whenever the test code invokes a method of the code under test, it exposes a part of its execution state to the code under test. We define the *prestate-footprint* of a method invocation as the subset of memory locations that is reachable from the arguments and the receiving object. During the execution of method invocation, the footprint may change. When the method returns (either exceptionally or normally), the *poststate-footprint* is the subset of memory locations that is reachable from the original arguments.

After the method invocation has returned, the test suite needs to validate whether the execution state is correct using assertions. A memory location is considered validated when it is read during the evaluation of the condition of the assertion.

Then lightweight state coverage is defined as the ratio of the post-state footprints of all method invocations from the test suite to the code under test which are validated using assertions.

B. Algorithm

In this section, we present an algorithm to compute lightweight state coverageṠince lightweight state coverage is a highly dynamic metrics, it is computed by monitoring the execution of the test suite at runtime. At load time, the test suite and the code under test are transformed in order to insert callbacks to a monitor. During the execution, this monitor captures information that allows the computation of lightweight state coverage.

Essentially, the algorithm is split into two phases. First, the algorithm computes the poststate-footprint whenever a method invocation returns from the code under test to the test suite. When the code under test is loaded, all publicly accessible methods are transformed according to the schema described in Figure 2. When entering the method, the instrumentation uses the monitor to checks whether the execution is already in the code under test. If not, all arguments that are object are registered with the monitor. In the example, argument $a1$ and $a3$ are objects, but $a2$ is not (e.g. an $integer$ or $boolean$). The body of the method is executed in a try-finally construction. This ensures that the snapshot method can compute the monitor poststate footprint when the method returns. In addition, the monitor is notified that the execution leaves the code under test.

Second, the code under test is transformed in order to track whether the execution is evaluating the condition of an assertion and when this evaluation reads object fields. Figure 3 shows conceptually what this instrumentation looks like. Before and after invoking an assertion method, the monitor is notified that execution enters or leaves an assertion. In addition, all methods are instrumented to notify the monitor when the execution reads an object field. In practice, the condition need not be in a separate method. In addition, we assume that the conditions of assertions are observationally pure, which means they have no observable side effects. For observationally pure methods, it is not necessary to compute the poststate-footprint. This allows one to use getters and invariant methods in the conditions of assertions, which enable checking inaccessible state.

```
class A {
  public Tr m(T1 a1, T2 a2, ..., Tn an) {
    Body
  }
}

class A' {
  public Tr m(T1 a1, T2 a2, ..., Tn an) {
    boolean entering = Monitor.enter();
    if(entering) {
      Monitor.register(a1);
      Monitor.register(a3);
      ...
    }
    try {
      Body
    } finally {
      if(entering) {
        Monitor.snaphot();
        Monitor.exit();
      }
    }
  }
}
```

Fig. 2. Footprint-transformation

```
class Test {
  public void m() {
    ...
    Assert.isTrue(C);
  }
}

class Test {
  public void m() {
    ...
    Monitor.enterAssert();
    Assert.isTrue(C(...));
    Monitor.leaveAssert();
  }
  boolean C(...) {
    ...
    Monitor.readField(o, f);
    a = o.f;
    ...
  }
}
```

Fig. 3. Assertion transformation

Finally, Figure 4 shows the pseudocode for the monitor which computes state coverage at runtime. The field $inCodeUnderTest$ tracks whether execution is in the domain under test, and it is updated by the methods $enter$ and $exit$. Similarly, the methods $enterAssert$ and $leaveAssert$ use the field $inAssert$ to track whether execution is in an assertion. The method $register$ stores all arguments in the $args$ queue, Next, the method $collect$ computes the least set

```
class Monitor {
  boolean inCodeUnderTest = false;
  boolean inAssert = false;

  int counter = 0;
  Queue args = new Queue();
  Set<Triple> locs = new Set<Triple>();
  Set<Triple> valLocs = new Set<Triple>();

  boolean enter() {
    boolean result = !inCodeUnderTest;
    inCodeUnderTest = true;
    return result;
  }
  void exit() {
    inCodeUnderTest = false;
  }
  void enterAssert() {
    inAssert = true;
  }
  void leaveAssert() {
    inAssert = false;
  }
  void register(Object o) {
    args.push(o);
  }
  void snaphot() {
    Set seen = new Set();
    while(!args.size()>0) {
      Obj o = args.pop();
      if(!seen.contains(o)) {
        seen.add(o);
        for(Field f:o.getFields()) {
          locs.add(new Triple(counter, o, f));
          if(f.hasObjectType()) {
            args.push(o.f);
          }
        }
      }
    }
    counter++;
  }
  void readField(Object o, Field f) {
    if(inAssert && counter > 0) {
      t = new Triple(counter - 1, o, f);
      if(locs.contains(t))
        valLocs.add(t);
    }
  }
  double getLWS() {
    return (double)valLocs.size()/locs.size();
  }
}
```

Fig. 4. Runtime monitor

of objects that contains the arguments and is closed under reachability and registers all corresponding memory locations in the set $locs$. Then, when a $field$ of an object is read in an assertion, the $readField$ method marks the corresponding memory location as validated by adding it to the set $valLocs$. Finally, lightweight state coverage is the size of $valLocs$ divided by the size of $locs$.

IV. Preliminary evaluation

In this section, we perform a preliminary evaluation of lightweight state coverage. First, we describe how we implemented the algorithm in the previous section. Then, we describe the performance of this prototype on the Apache Commons Collections library. Finally, we discuss some of the design decisions and give some alternatives.

A. Implementation

We have created a prototype implementation of lightweight state coverage for the Java Virtual Machine. Unlike the presentation of the algorithm III-B, this prototype works directly on Java Bytecode. We use the java-agent infrastructure and the Apache Bytecode Engineering Library [10] to implement the transformation algorithms.

B. Performance

One of the reasons to approximate state coverage is to improve the efficiency and thus enable the usage of state coverage on large systems in the standard build cycle.

Koster [3] reports that state coverage is about 70 times slower than the standard JUnit test runner. Only about half of this overhead is due to the taint analysis.

To evaluate the performance of lightweight state coverage, we have measured the execution time of our prototype on the Apache Commons Collections library [9], which offers many powerful data structures to accelerate development of the most significant Java applications. Table I shows the runtime overhead for tracking lightweight state coverage. While the overhead is still significant, it between 2 and 10 times the execution time of the original test suite. This is within the acceptable range for executing a test suite offline. In addition, there is still room for optimization of the prototype.

Finally, the tests that have the biggest slowdown are often not the best unit tests. For example, the method $PriorityBufferTest.testRandom$ randomly creates priority buffers and checks whether they are correctly sorted. This process is repeated 500 times. Therefore, since the total number of memory locations grows large, it is not surprising that lightweight state coverage causes a high overhead.

C. Discussion

Unlike [4], [5], lightweight state coverage does not track the modified memory location throughout the execution of the code under test. This has several advantages: First, when the code under test is not completely available (e.g. because it running on a remote server or is native code), it is often hard to monitor its execution. Since state coverage only collects the state which must be validated at the moment the code under test returns, this code is always available. Second, even if the code is available, typical programs usually allocate some intermediate data structures to satisfy their requirements. When the execution returns to the client, this intermediate state is of no interest to the client and becomes unreachable. Therefore, it is impossible to validate this state. When tracking throughout the execution of the program, it is necessary to prune these unreachable memory locations.

Since lightweight state coverage no longer uses dependency tracking, the approximation of which memory locations influence an assertion is less precise, i.e. the result may include too much memory locations. It is easy to write programs that read too much state. However, these are often bad programs. This is not different when using information flow analysis. We rely on the developers to weed out malicious attempts to manipulate the metric. Other metrics such as statement coverage can in principle also be manipulated.

	With lightweight state coverage	Original	Percentage
org.apache.commons.collections.test	1,248	0,4773	261,47%
org.apache.commons.collections.test.bag	0,533	0,157	339,49%
org.apache.commons.collections.test.bidimap	28,573	2,928	975,85%
org.apache.commons.collections.test.buffer	42,084	8,201	513,16%
org.apache.commons.collections.test.collection	0,597	0,153	390,20%
org.apache.commons.collections.test.comparators	0,332	0,136	244,12%
org.apache.commons.collections.test.comparators.sequence	1,988	0,484	410,74%
org.apache.commons.collections.test.functors	0,157	0,063	249,21%
org.apache.commons.collections.test.iterators	0,854	0,239	357,32%
org.apache.commons.collections.test.keyvalue	0,092	0,051	180,39%
org.apache.commons.collections.test.list	2,554	1,017	251,13%
org.apache.commons.collections.test.map	11,568	6,881	168,12%
org.apache.commons.collections.test.set	1,738	0,613	283,52%
org.apache.commons.collections.test.splitmap	0,15	0,032	468,75%
org.apache.commons.collections.test.trie	1,235	0,46	268,48%
total	93,703	21,8923	428,02%

TABLE I. PERFORMANCE OF LIGHTWEIGHT STATE COVERAGE ON COMMONS-COLLECTIONS

Lightweight state coverage assumes observational purity of assertion bodies. When a method is called inside an assertion, this method may also change the internal state of the application. If the internal state of an application can only be observed using methods that have side effects, it may be impossible to create a test suite that has full lightweight state coverage. Since well-established metrics suffer from similar effects (e.g. statement coverage in the presence of dead code), this is not necessarily a problem. In addition, it is best practice to provide a sufficiently large set of observationally pure methods to inspect the internal state of the application (either as getters, or as invariants).

V. RELATED WORK

State coverage is a *test adequacy metric* (See Zhu et al. [1] for a survey on test adequacy criteria). Structural coverage metrics (such as statement coverage) are most popular in this area. They all measure to some extent the degree of exploration, an abstraction of the subset of the execution paths of the program that are exercised by the test suite. State coverage is orthogonal to these metrics, since it measures the strength of the test oracle. Therefore, state coverage is most powerful in combination with the existing approaches.

State coverage is most closely related with all-defs [11] coverage. The critical difference between both is that dataflow coverage works with all state reads, whereas state coverage focuses on state reads that influence the result of an assertion. This difference makes state coverage measure the strength of the assertions in a code base, instead of measuring whether the code base is sufficiently explored.

The essential difference between lightweight state coverage and classical state coverage [2]–[4] is in the way they measure whether a state update influences the result of an assertion. Classical state coverage uses a complicated dependency analysis, while lightweight state coverage uses a simple read-write analysis. In addition, lightweight state coverage requires all reachable state to be checked in an assertion rather than all updated state. Throughout the paper, we have discussed an object-sensitive version of state coverage [4]. Generalizing the lightweight approach to object-insensitive state coverage is trivial.

Next, fault-based test adequacy criteria (mostly mutation testing [12]) measure the fault finding capability of a test suite. Unlike existing structural test adequacy criteria, mutation testing can be used to evaluate the strength of the test oracle. Mutation testing injects faults into the codebase and checks whether the test suite can observe the injected fault. Often mutation testing requires generating and executing millions of mutants. The mutation adequacy score divides the amount of killed mutants by the amount of non-equivalent mutants. Unfortunately, deciding whether a mutation is equivalent is undecidable in general, and therefore often requires human interaction. State coverage achieves some of the benefits of mutation testing, without the performance overhead and complexity of mutation testing.

VI. CONCLUSION

In this paper, we have defined Lightweight state coverage an approximation of state coverage that only requires a simple read analysis. In addition, all reachable state which is shared between the application and the code under test must be checked using assertions. As a result, lightweight state coverage is significantly faster than the original approach using dependency tracking. In addition, lightweight state coverage may potentially indicate errors of omission. Finally, lightweight state coverage is much less sensitive to unavailable code and unreachable state.

In future work, we plan to asses whether these improvements also cause a better correlation with the number of defected detects.

ACKNOWLEDGMENT

Dries Vanoverberghe is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (FWO). This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the IWT, and by the Research Fund K.U.Leuven.

REFERENCES

[1] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, pp. 366–427, December 1997. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[2] K. Koster and D. Kao, "State coverage: a structural test adequacy criterion for behavior checking," in *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ser. ESEC-FSE companion '07. New York, NY, USA: ACM, 2007, pp. 541–544. [Online]. Available: http://doi.acm.org/10.1145/1295014.1295036

[3] K. Koster, "A state coverage tool for junit," in *Companion of the 30th international conference on Software engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 965–966. [Online]. Available: http://doi.acm.org/10.1145/1370175.1370210

[4] D. Vanoverberghe, J. de Halleux, N. Tillmann, and F. Piessens, "State coverage: software validation metrics beyond code coverage," in *Proceedings of the 38th international conference on Current Trends in Theory and Practice of Computer Science*, ser. SOFSEM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 542–553. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27660-6_44

[5] D. Vanoverberghe, E. Eyckmans, and F. Piessens, "State coverage: An empirical analysis based on a user study," in *SOFSEM 2013: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, P. Emde Boas, F. Groen, G. Italiano, J. Nawrocki, and H. Sack, Eds. Springer Berlin Heidelberg, 2013, vol. 7741, pp. 469–480. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35843-2_40

[6] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *J. Syst. Softw.*, vol. 4, no. 4, pp. 289–300, Nov. 1984. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(84)90028-1

[7] R. L. Glass, "Persistent software errors," *IEEE Trans. Softw. Eng.*, vol. 7, no. 2, pp. 162–168, Mar. 1981. [Online]. Available: http://dx.doi.org/10.1109/TSE.1981.230831

[8] ——, "Loyal opposition - frequently forgotten fundamental facts about software engineering," *IEEE Software*, vol. 18, no. 3, 2001.

[9] The Apache Software Foundation, "Commons collections," http://commons.apache.org/proper/commons-collections/.

[10] M. Dahm, "Byte code engineering with the bcel api," Tech. Rep., 2001.

[11] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Softw. Eng.*, vol. 11, pp. 367–375, April 1985. [Online]. Available: http://dx.doi.org/10.1109/TSE.1985.232226

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.